perplexity

# explain the 2nd code present in the file

The **second code** in the file is a **recursive implementation of Breadth-First Search (BFS) to find a path from a start vertex to a goal vertex in an undirected graph**. Below is a detailed line-by-line explanation of this code and its output.

### Line-by-Line Explanation of the Recursive BFS Code

```
from collections import defaultdict, deque
```

- Imports `defaultdict` to create a graph with default empty sets for neighbors.
- Imports `deque` (double-ended queue) for efficient FIFO queue operations used in BFS.

```
def bfs_recursive_path(graph, queue, visited, goal):
```

- Defines a recursive function `bfs_recursive_path` that takes:
  - `graph`: the graph dictionary,
  - `queue`: a deque holding tuples `(vertex, path_to_vertex)`,
  - `visited`: a set of visited vertices,
  - `goal`: the target vertex to find.

```
    if not queue:
        print("Goal not found")
        return None
```

- Base case: if the queue is empty and goal not found, print "Goal not found" and return `None`.

```
    vertex, path = queue.popleft()
```

- Pops the first element from the queue (FIFO order).
- `vertex` is the current vertex being explored.
- `path` is the list of vertices from the start to this vertex.

```
    if vertex == goal:
        print(f"Path to goal {goal}: {path}")
        return path
```

- If the current vertex is the goal, print the path found and return it, ending recursion.

```
    visited.add(vertex)
```

- Mark the current vertex as visited to avoid revisiting.

```
    for neighbor in graph[vertex]:
        if neighbor not in visited and all(neighbor != n[^0] for n in queue):
            queue.append((neighbor, path + [neighbor]))
```

- For each neighbor of the current vertex:
  - Check if it is not visited.
  - Also check if it is not already in the queue (to avoid duplicates).
  - If both conditions hold, append the neighbor and the updated path (current path + neighbor) to the queue.

```
    return bfs_recursive_path(graph, queue, visited, goal)
```

- Recursively call `bfs_recursive_path` with the updated queue and visited set.

### Graph Input and Setup

```
graph = defaultdict(set)
num_vertices = int(input("Enter the number of vertices: "))
```

- Creates an empty undirected graph.
- Takes the number of vertices as input.

```
for _ in range(num_vertices):
    vertex = input("Enter a vertex: ")
    neighbors = input("Enter its neighbors (separated by spaces): ").split()
    graph[vertex].update(neighbors)
    for neighbor in neighbors:
        graph[neighbor].add(vertex)  # Ensure undirected connection
```

- For each vertex, input its neighbors.
- Update the graph to add neighbors.
- Also add the vertex to each neighbor's adjacency list to make the graph undirected.

```
start_vertex = input("Enter the starting vertex: ")
goal_vertex = input("Enter the goal vertex: ")
```

- Inputs the start and goal vertices for the BFS search.

```
print("\nRecursive BFS traversal (with path to goal):")
visited = set()
queue = deque([(start_vertex, [start_vertex])])
bfs_recursive_path(graph, queue, visited, goal_vertex)
```

- Prints a header.
- Initializes an empty set `visited`.
- Initializes the queue with a tuple of the start vertex and the path containing only the start vertex.
- Calls the recursive BFS function to find and print the path.

### Sample Input and Output Example

**Input:**

```
Enter the number of vertices: 5
Enter a vertex: a
Enter its neighbors (separated by spaces): b c e
Enter a vertex: b
Enter its neighbors (separated by spaces): a d
Enter a vertex: c
Enter its neighbors (separated by spaces): a d
Enter a vertex: d
Enter its neighbors (separated by spaces): a b c
Enter a vertex: e
Enter its neighbors (separated by spaces): a d
Enter the starting vertex: a
Enter the goal vertex: c
```

**Output:**

```
Recursive BFS traversal (with path to goal):
Path to goal c: ['a', 'c']
```

### What Happens Here?

- The graph is undirected with edges connecting the vertices as per input.
- The recursive BFS starts at vertex `a`.
- It explores neighbors of `a`: `b`, `c`, and `e`.
- Since `c` is a direct neighbor of `a`, the path `['a', 'c']` is found immediately.
- The function prints the path and returns it, ending the recursion.

### Summary

- This code implements BFS **recursively** to find the shortest path from a start vertex to a goal vertex.
- It uses a queue (`deque`) to keep track of vertices to visit along with the path taken so far.
- The recursion continues until the queue is empty or the goal is found.
- The graph is undirected, ensured by adding edges in both directions.
- The output shows the shortest path found by BFS.

This explanation covers the purpose, working, and output of the second code in the file clearly and thoroughly.

⁂